一种符号执行制导的循环内界分析方法

赵祖威[1,2],冯世宁[3,4],汤恩义[1,2],陈 鑫[1],李宣东[1,2],潘敏学[1,2],赵 晨[1,2]

(1.南京大学软件新技术国家重点实验室,江苏南京 210023;2.南京大学软件学院,江苏南京 210093;
3.南瑞集团公司(国网电力科学研究院),江苏南京 211106;4.国电南瑞科技股份有限公司,江苏南京 211106)

摘 要: 循环是计算机中重要的复杂程序结构.很多应用场景要求静态分析循环可能达到的最大迭代次数,即循环边界(Loop Bound).对应技术在文献中被称为循环边界分析(Loop Bound Analysis).现有的循环边界分析均使用保守方式进行外界分析,即产生略高于循环边界的近似值.基于这一现状,本文提出了一种自动地循环内界分析方法,产生略低于循环边界的近似值.当用户综合利用外界与内界分析,能将循环边界值约束到一个统计区间,从而能对分析结果获得更为完整的认识.本文基于循环条件制导的符号执行(Symbolic Execution)技术实现了循环内界分析,该技术的本质在于它能够利用符号执行符号化推导程序执行约束的特点,准确求解循环在程序所有合法输入条件下的边界值,并由生成的测试用例来保证该边界值一定可达(即保证是循环内界).本文对符号执行制导技术进行了优化,并在多组已有研究采用的基准用例集上进行了实例评估,实验结果表明,本文的循环内界分析方法具备准确性和高效性,可以满足应用需求.

关键词: 循环边界分析;符号执行;软件测试

中图分类号: TP311.5 **文献标识码:** A **文章编号:** 0372-2112 (2017)11-2582-11

电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2017.11.003

A Symbolic Execution Guided Inner Loop Bound Analysis

ZHAO Zu-wei[1,2],FENG Shi-ning[3,4],TANG En-yi[1,2],
CHEN Xin[1],LI Xuan-dong[1,2],PAN Min-xue[1,2],ZHAO Chen[1,2]

(1. *State Key Laboratory for Novel Software Technology*,*Nanjing University*,*Nanjing*,*Jiangsu* 210023,*China*;
2. *Software Institute*,*Nanjing University*,*Nanjing*,*Jiangsu* 210093,*China*;
3. *NARI Group Corporation* (*State Grid Electric Power Research Institute*),*Nanjing*,*Jiangsu* 211106,*China*;
4. *NARI Technology Co.*,*Ltd.*,*Nanjing*,*Jiangsu* 211106)

Abstract: Loop is an important program structure in computer. Many applications need to estimate the maximum iteration number of loops in programs by loop bound analysis. Existing loop bound analysis uses conservative methods to derive outer loop bounds,which estimates the bounds higher than the real ones. In this paper,we propose an automatic inner bound analysis,which generates bounds slightly lower than the real ones. When users combine the inner bound analysis with traditional outer bound analysis,they can restrict every real loop bound in an interval and get more information about the loops. We implement the inner bound analysis by a scope-condition guided symbolic execution. The insight of our technique is that when symbolic execution substitutes program inputs by symbols in its derivation,it generates loop bounds for all valid inputs and generates corresponding test cases that make the inner bounds feasible. We optimize the technique and evaluate it on several benchmarks. The results show that the analysis is precise and efficient.

Key words: loop bound analysis;symbolic execution;software testing

收稿日期:2016-07-25;修回日期:2016-12-16;责任编辑:马兰英
基金项目:国家自然科学基金(No. 61402222,No. 61632015);国家重点研发计划(No. 2016YFB1000802);教育部高等学校博士学科点专项科研基金(No. 20110091120058);江苏省产学研项目(No. BY2014126-03)

1　引言

循环是计算机中重要的复杂程序结构. 许多应用场景要求静态分析循环可能达到的最大迭代次数,文献中称之为循环边界(Loop Bound),而对应的计算方法被称为循环边界分析(Loop Bound Analysis)[1~3]. 循环边界的使用场景有:在程序优化中作为各代码段对全局性能影响的依据;在程序影响分析中用于进一步分析循环中代码的影响范围;在实时系统分析中用于估测系统各任务的最坏情况执行时间;检测触发条件较为苛刻、需要深入循环才会发生的软件缺陷等等.

现有的循环边界分析方法均为外界分析[1,4],这些方法采用较保守的策略,在不能准确分析时保证分析值一定大于实际循环边界值,因此我们称这样的分析值为循环外界(Outer Bound). 这类外界分析方法特别适用于保证实时任务安全性的分析场景,即保证实时系统中循环的实际最大迭代次数一定不超出分析值. 然而,外界分析方法并不能测算当前的分析值离实际的边界值有多远,当外界分析方法不够准确时,会因为系统预留了过多的计算资源而造成浪费[2~4]. 基于这一现状,本文提出一种循环内界的自动分析方法(Inner Bound Analysis). 该方法尽可能准确地获得循环实际能达到的边界值,即注重边界值的可达性. 在不能准确分析循环边界的条件下,内界分析得到略小于实际边界的分析值. 该方法的意义在于,当用户综合利用外界分析和内界分析时,可以获得实际边界的范围区间,从而对分析结果有更为完整的认识. 例如,用户可以明确某循环边界的具体范围介于内界 495 和外界 511 之间,并获得该循环达到 495 次迭代的测试用例.

本文基于循环条件制导的符号执行(Symbolic Execution)技术来实现循环内界分析. 符号执行技术[5~8]将程序输入定义成可代表任意值的符号,并沿程序的控制流推导各中间变量的关系性质. 当程序中存在多条路径时,符号执行会创建多个执行状态来追踪各路径下的变量关系,并记录各路径的条件约束. 最终,利用约束求解器(Constraint Solver)来获得覆盖特定路径的测试输入. 本文通过制导符号执行过程使其能够高效地推导各循环达到最大迭代目标路径的约束条件,从而实现循环内界的自动分析. 同样,我们也能生成该约束条件对应的测试用例,以覆盖循环的最大迭代路径,供进一步分析与测试.

本文基于开源符号执行平台 KLEE[9] 实现了自动循环内界分析,并在多组已有研究采用的基准用例集上进行了实例评估. 实验结果显示本文的循环内界分析方法是准确和高效的,能满足大多数的应用需求.

2　符号执行

符号执行(Symbolic Execution)是一项近年来广受关注的[7~14]软件测试自动生成技术,它的基本思想是:将程序输入定义成表示任意值的符号,并由此推导程序各中间数据和执行结果. 因此,各变量值在符号执行中均被表示成了输入符号的表达式. 符号执行通过创建执行状态来记录程序中不同执行路径下各变量的符号表达式. 当符号推导遇到一个条件分支时,符号执行平台会从当前执行状态中复制出一个新的执行状态,并在新状态中记录条件为假的分支条件约束,而当前的执行状态会被记录为条件为真的分支条件约束而继续推导. 最终当某一执行状态的推导到达程序出口时,符号执行平台通过约束求解器求解当前状态的总条件约束,并生成覆盖对应路径的测试用例. 本节通过一个具体的例子来说明这一过程,如图 1 所示.

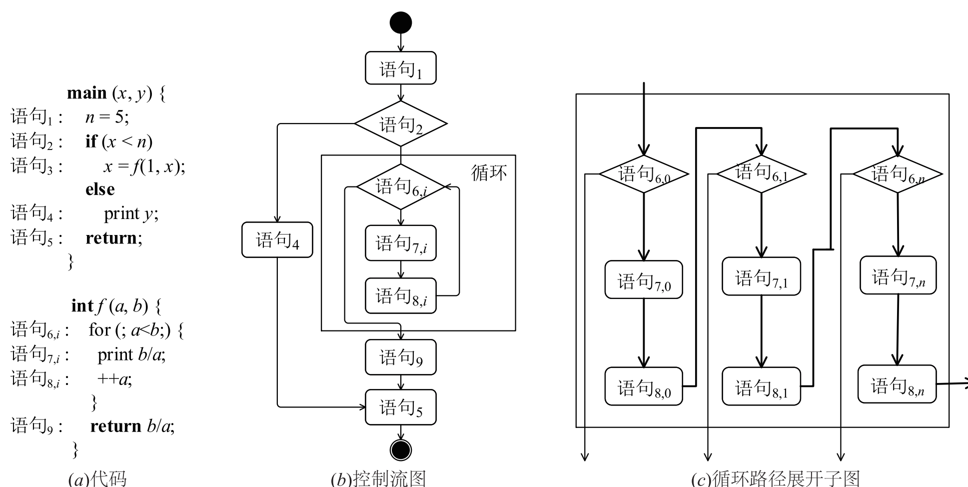

图1　循环程序示例

图1(a)给出了一段带循环的示例程序,它的控制流(Control Flow)如图1(b)所示. 在图1(a)中main函数的入口处,符号执行平台将输入变量x和y表示为可代表任意值的符号,并建立符号执行的第一个执行状态s_0. 符号执行中每个执行状态s由5个信息域(pc, cv, sv, ex, c)组成,具体包括:(1)程序计数器$s.pc$,记录了执行状态s的当前执行位置;(2)所有非符号变量的具体值$s.cv$,记录了各具体变量到值的映射,例如图1(a)中变量n被赋值为5;(3)程序中的符号值$s.sv$,记录了符号变量到符号表达式的映射,例如示例程序中的变量x和y在程序入口处被初始化,映射为表示任意值的符号;(4)符号操作表达式$s.ex$,记录程序中符号变量间的关系;(5)当前路径的符号约束$s.c$,记录覆盖当前路径须满足的条件,例如示例程序中的$x < n$. 对于每一个执行状态s,符号执行由$s.pc$获得该状态的下一条指令,并由该指令推导更新s中其它各信息域的值. King等人的工作[6]定义了这一推导过程,在本文中,我们将其封装为s的成员函数s.symbolicExecution( ),它的返回值为经过当前指令更新的状态集. 特别地,当s在推导过程中遇到条件分支(例如if b then S else S')时,它会产生含有不同路径约束的新状态,返回状态集$\{s_t, s_f\}$. 这里状态s_t和s_f分别记录了true分支和false分支对应的执行状态,即$s_t.c \leftarrow s.c \wedge b$和$s_f.c \leftarrow s.c \wedge \neg b$,而$s_t$和$s_f$中的其他信息皆从原状态$s$中拷贝. 为节省内存,本文方法将状态$s_t$各信息域直接更新到原状态$s$里而仅新建状态$s_f$,我们也称新建的状态为$s'$,从而把这一过程定义为$\{s, s'\} \leftarrow s$.forkExecution( ). 因此,仅当符号执行遇到条件分支时执行状态的数量才会增加.

程序中存在循环时,符号执行会导致状态爆炸[7]. 因为默认的符号执行算法将循环的各次迭代完全展开,而实际程序中单个循环的展开路径就可能有无穷多条. 更进一步,即使程序中每个循环的展开路径为有限条,多个循环的组合情况也使得展开的路径总数呈指数级增长. 图1(c)展示了图1(a)中循环在路径展开下的符号执行控制流. 由于循环的每次迭代都会使符号推导遇到条件分支,因此覆盖循环中所有的展开路径会十分困难. 所幸的是,循环内界分析并不需要完整覆盖所有的展开路径,而仅需覆盖使循环达到最大迭代次数的目标路径(图1(c)中加粗的路径). 因此,循环内界分析算法的核心在于高效地制导符号执行覆盖循环的目标路径,并产生对应的路径约束. 通过约束求解,即可获得循环内界与覆盖该循环内界路径的测试输入.

3 循环内界分析算法

我们设计了一个两阶段的符号执行制导算法来完成各循环的内界分析. 在阶段1,算法会在循环外进行推导,这时符号执行平台会从程序入口开始尽可能覆盖每一条路径来收集程序中各循环的前置约束条件. 紧接着分析算法会切换到阶段2,在这一阶段每一个执行状态会制导分析深入循环的目标路径,并由此推导该目标路径的约束条件,并生成相应的循环内界和测试用例. 下面我们分阶段介绍算法的执行过程.

3.1 阶段1

为制导符号执行生成循环内界,本文对状态s进行了扩展,增加了一个信息域ζ以记录循环的上下文信息. ζ是一个对应于嵌套循环每一层的栈,当状态s到达循环入口时,它会将该循环的上下文信息loop_info压入栈$s.\zeta$中. 每一层循环的上下文信息loop_info包含一个循环标签loop_label,和该循环目前的迭代计数. 当执行状态s离开循环时,算法会使$s.\zeta$出栈,并根据出栈信息计算循环内界.

算法1给出了循环内界分析的主流程,也是阶段1的主要执行过程. 这一流程主要通过两个存放执行状态的全局队列Q_1、Q_2以及一个记录循环与执行状态对应关系的映射Ψ来完成的. 其中Q_1记录了应该由阶段1处理的执行状态,Q_2记录了需要由阶段2处理的执行状态,映射Ψ将每一个循环映射到一对状态集<in, out>上,其中集合in存放了已经进入该循环的程序执行状态,而集合out存放将要离开该循环的执行状态.

算法1 循环内界分析主函数

```
输入:ExecState initialState;
输出:Map<Loop,int> Bounds;       //各循环内界
     Map<Loop,Case> TestCases;  //覆盖内界的测试用例
初始化:Queue<ExecState> Q1,Q2;
      class PairSet{ Set<ExecState> in,out;}
      Map<Loop,PairSet>Ψ;
1:  Q1.enqueue(initialState);
2:  while Q1.size>0 do
3:    ExecState s←Q1.dequeue();
4:    while s.pc ! = LoopEntry && s.pc! = EXIT
5:          && ! TIMEOUT do
6:      states←s.symbolicExecution(); //可能会生成新状态
7:       ∀s' ∈ states ∧ s' ≠ s, Q1.enqueue(s');
8:    end while
9:    if s.pc == EXIT then
10:      delete s;
11:   else if s.pc == LoopEntry then
12:      loop_info ← <loop_label,0>
13:      s.ζ.push(loop_info);
14:      Q2.enqueue(s);
15:   else
16:      Q1.enqueue(s);       //符号s仍在阶段1执行
```

```
17:     end if
18:     if Q1.empty() || TIMEOUT then
19:        loopinit();
20:        boundCompute();       //切换到阶段 2
21:     end if
22:  end while
```

算法 1 以符号执行平台进入程序的第一个状态 initialState 为输入,输出循环内界和对应的测试用例. 算法开始时,Q_1 中只有状态 initialState 而 Q_2 为空. 算法 1 每次从 Q_1 中取一个状态 s 进行推导,未遇到循环入口时,推导过程即为传统的符号执行过程(第 6 行). 而当遇到循环入口时,s 会从 Q_1 被移动到队列 Q_2 中,以便在阶段 2 中进一步处理(第 11 行). 这一过程会更新状态 s 的循环上下文信息(11 ~ 14 行),将循环标签和初始计数 0 压入信息栈 ζ 中. 随着算法的推进,当 Q_1 中所有状态均被处理完或超时的情况下,算法会调用 loopInit( ) 和 boundCompute( )(19 ~ 20 行)切换到阶段 2. loopInit 函数主要为切换做准备工作,而 boundCompute 函数会切换到阶段 2 来推导循环内界. 在完成各状态在循环内的推导后,boundCompute 函数返回,此时算法会切回阶段 1 并继续在循环外推导各执行状态,以便进一步处理程序中各执行状态尚未遇到的循环.

以图 1(a)的代码为例,符号执行开始时创建初始状态 s_0,并加入到队列 Q_1 中. 而后算法 1 从 Q_1 中取出 s_0 进行推导,调用 symbolicExecution 更新 s_0 中变量 n 的值为 5,并在图 1(a)的语句 2 处产生一个新状态 s_1 进入 else 分支,而状态 s_0 的条件约束 $s_0.c$ 被更新为 $x<5$. 符号执行在继续推导状态 s_0 时会进入图 1(a)的函数 $f(1,x)$ 内,并在第一次到达语句 6 时,由算法 1 的第 11 行判断到达循环入口,根据算法 1 的流程,$s_0.\zeta$ 会记录循环标号 l,且 s_0 会被加入到 Q_2 中. 此时算法继续推导 Q_1 中的状态 s_1,s_1 在后续执行中未遇到循环而被释放,因此循环 l 的前置条件已完全被记录在 s_0 的路径约束中. 此时算法 1 会进入阶段 2 来分析这一循环.

3.2　阶段 2

上一节中,算法 1 通过调用函数 loopInit( ) 和 boundCompute( )来初始化并将程序切换到阶段 2,以便对每个循环做具体的内界分析,本节描述这些函数在阶段 2 的处理过程. 算法 2 描述了函数 loopInit 初始化切换阶段 2 的过程,这一过程首先初始化全局映射 Ψ,使每一个未完成映射的循环映射到空的执行状态集 in 和 out 上(1 ~ 4 行);然后通过调用 stateSetMerge 对队列 Q_2 做执行状态的合并与优化(第 5 行);最终,通过读取队列 Q_2 中各执行状态的循环上下文信息栈来更新映射 Ψ,使其将循环映射到准确的执行状态集上.

算法 2　切换到阶段 2 的初始化

```
Begin loopInit
1:  PairSet empty_pair = {in←∅, out←∅}
2:  ∀ loop_label, Ψ(loop_label) == NIL do
3:     Ψ← <loop_label, empty_pair>;
4:  end for
5:  stateSetMerge(Q2);
6:  ∀ s ∈ Q2 do
7:     Ψ(s.ζ.peek().loop).in.add(s);
8:  end for
End loopInit
```

算法 3 给出了阶段 2 的循环内界推导过程,即函数 boundCompute 的执行过程. 算法 3 由 Q_2 中的执行状态开始,制导产生循环最深路径上的执行状态及其约束条件. 由于在循环入口和出口处分别对各状态进行了优化合并,使得符号执行可以把更多的计算资源集中在循环中迭代次数最多的目标路径上. 为了更直观的说明算法 3 在推导过程中的处理指令,我们先由图 2 来介绍算法 3 处理的 4 个指令类别,分别为:循环入口条件;嵌套循环入口条件;循环出口分支;循环内分支. 循环入口条件是判定循环又将开始新一轮迭代的程序位置;对于嵌套循环来说,内层循环也会有嵌套循环入口条件;由于循环可能有多个出口,循环出口分支指决定程序控制流将离开循环的最后一个分支条件;除此之外,循环内的其它分支条件我们统称为循环内分支. 在本文的算法实现中,我们基于已编译产生的二进制代码对图 2 中涉及的相关指令进行处理,这 4 类指令均在二进制码上进行了标注,因此本文的处理方法理论上并不依赖于源代码的语言和语法结构.

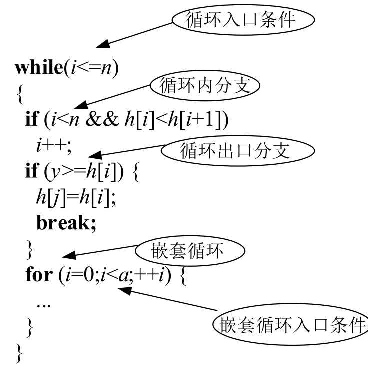

图2　一个展示阶段2算法中术语的循环示例

算法 3 每次从 Q_2 中取一个状态 s,沿循环的控制流进行符号执行推导,其推导重点在对遇到的 4 类指令进行特殊处理. 循环入口条件在本算法中起关键作用,它是判定循环内界路径的决定性因素. 当状态 s 遇

到循环入口条件时(4～11 行),算法将迭代计数加 1,并由 forkExecution 获得下一轮迭代的约束,并将该约束保存在 s 中,而离开循环的约束则被保存在状态 s' 中. 在更新映射 Ψ 后,算法调用 loopExitDec 来判断是否已到达循环内界,如果到达则收集相应的信息,否则继续迭代. 当 s 遇到嵌套循环入口条件时(12～15 行),算法将内层循环的上下文压栈,并制导 s 优先获得内层循环内界. 当 s 遇到循环出口分支时(16～28 行),算法首先判别由出口分支离开循环的状态 s_o,更新映射 Ψ,并由 loopExitDec 做循环出口推导处理. 对于其它循环内分支(29～31 行),算法将分支的产生状态统一记录在 Ψ 的当前循环内,以便在算法推导中统一处理.

算法 3 循环内界推导

```
Begin boundCompute
1:  while Q2.size > 0 do
2:    ExecState s←Q2.dequeue();
3:    while true do
4:      if s.pc == currentLoopEntry          //处理循环入口条件
5:        s.ζ.peek().count++;
6:        {s,s'}←s.forkExecution();
7:        ∀loop_info∈s'.ζ ∧ loop_info≠ s'.ζ.peek(), Ψ(loop_
          info.loop).in.add(s');
8:        Ψ(s'.ζ.peek().loop).out.add(s');
9:        if loopExitDec(s.ζ.peek().loop) == true
10:         break;
11:       end if
12:     else if s.pc == nestedLoopEntry   //处理嵌套循环入口条件
13:       nested_loopinfo←<nested_looplabel,0>
14:       s.ζ.push(nested_loopinfo);
15:       Ψ(nested_looplabel).in.add(s);
16:     else if s.pc == exitFork             //处理循环出口分支
17:       {s,s'} = s.forkExecution();
18:       ∀ loop_info∈s'.ζ, Ψ(loop_info.loop).in.add(s');
19:       if s is at the exit branch then
20:         ExecState s_i←s', s_o←s;
21:       else
22:         ExecState s_i←s, s_o←s';
23:       end if
24:       Ψ(s_o.ζ.peek().loop).in.remove(s_o);
25:       Ψ(s_o.ζ.peek().loop).out.add(s_o);
26:       if loopExitDec(s_o.ζ.peek().loop) == true
27:         break;
28:       end if
29:     else if s.pc == Fork                 //处理循环内分支
30:       {s,s'} = s.forkExecution();
31:       ∀ loop_info∈s'.ζ, Ψ(loop_info.loop).in.add(s');
32:     else                        //其他指令按传统符号执行处理
33:       s.symbolicExecution();
34:     end if
35:   end while
36: end while
End boundCompute
```

算法 4 描述了循环出口处理函数 loopExitDec. 该函数首先清理循环内具有约束冲突的状态,如果经过清理后发现当前循环内的所有状态已被清理完毕,则意味着循环迭代次数已经到达循环内界,接着算法会调用 symbolicSetMerge 进行状态约束合并优化,记录计数器中的循环内界,并调用 generateTestCase 来求解当前状态的路径约束生成测试用例. 当算法判断最外层循环已处理完毕时,则返回 true 以便 boundCompute 进一步处理其它循环(第 22 行). 否则调用 symbolicSetMerge 进行状态约束的合并优化,并返回 false.

算法 4 循环出口处理

```
Begin loopExitDec(Loop loop)
9:  ∀s ∈Ψ(loop).in
10:   if solve(s.c) == mustBeFalse             //清理约束冲突
11:     Ψ(loop).in.remove(s);
12:   end if
13: end for
14: if Ψ(loop).in ==∅                //循环内所有状态处理完毕
15:   symbolicSetMerge(Ψ(loop).out);
16:   ∀s ∈Ψ(loop).out do
17:     bound = max(s.ζ.pop().count);
18:     if update(Bounds(loop),bound) then
19:       TestCases(loop)←generateTestCase(s.c);
20:     end if
21:   end for
22:   if ∀s ∈Ψ(loop).out,s.ζ.empty() == true   //最外层循环已
      处理完毕
23:     return true;
24:   end if
25: else
26:   symbolicSetMerge(Ψ(loop).in);            //如果当前循环内
      还有状态,则合并这些状态
27: end if
28: return false;
End loopExitDec
```

阶段 2 的算法也会进一步处理图 1(a)代码中语句 6～8 的循环 l,分析该循环的内界并产生对应的测试用例. 算法 2 首先对队列 Q_2 的各执行状态进行初始化,把循环 l 相关的执行状态加入到映射 Ψ 中. 经过初始化后 $\Psi(l)=<\{s_0\},\varnothing>$,且由于 Q_2 中仅含 s_0,无需对循环 l 的前置条件约束进行合并. 算法 3 负责 s_0 在进入循环 l 后的推导过程,由于该示例代码中没有嵌套循环、循环内分支和额外的循环出口,因此算法 3 仅需在每

一轮符号执行推导迭代时对循环的入口条件进行处理. 在第一次到达入口条件时,算法 3 的迭代计数 count 会加 1,并产生一个将会离开循环的新状态 s_2,且 $s_2.c$ 为 $x\leq 1$,而原状态 s_0 得以继续在循环中运行,其条件约束 $s_0.c$ 被更新为 $x<5\wedge x>1$,此时 $\Psi(l)$ 被算法 3 的 7 -8 行更新为 $<\{s_0\},\{s_2\}>$. 算法 3 在每一次处理入口条件后,会在第 9 行调用算法 4 判断当前循环是否处理结束,本次迭代中算法 4 直接返回 false. 下一次迭代时算法 3 会进一步产生新状态 s_3,且 $s_3.c$ 为 $x\leq 2$,而 $s_0.c$ 则被更新为 $x<5\wedge x>2$,$\Psi(l)$ 被进一步更新为 $<\{s_0\},\{s_2,s_3\}>$,此时算法 4 会在第 10 行判断 $s_0.c$ 是否存在解. 发现仍存在解时继续做下一次迭代,直到 3 次迭代之后 $s_0.c$ 成为 $x<5\wedge x>4$,算法 4 判断 $s_0.c$ 无解,则 s_0 被释放且循环 l 的分析结束,$\Psi(l)$ 中各状态的最大计数 3 会成为分析所得的循环内界,并对应地解得达到该循环内界的测试输入为 $x=4$. 为了高效地进行后续循环处理,算法 4 在第 15 行和第 26 行分别调用了 symbolicSetMerge 进行优化合并,具体的合并过程将在下一节中描述.

3.3　状态约束的优化合并

本文所述的循环内界分析算法会优先分析深入循环的路径,并获得循环内界和对应的测试用例. 然而未到达循环内界的各路径约束条件仍然是该循环后置条件的重要组成部分,并会进一步影响程序的后续路径推导. 由于程序逻辑的复杂性,未到达循环内界的条件约束数量往往很大,逐个推导会严重影响效率,因此本文设计了基于状态约束合并的优化算法(算法 5),使符号执行在处理循环后置条件的效率大大提高.

算法 5　执行状态及时合并函数

```
Begin symbolicSetMerge (Set<ExecState> S)    //状态集合并函数
1:  ∀ s_1, s_2 ∈ S
2:     s ← symbolicMerge(s_1, s_2);
3:     if s ≠ NULL
4:        S.remove(s_1); S.remove(s_2); S.add(s);
5:        replace(s_1, s); replace(s_2, s);
6:     end if
7:  end for
End symbolicSetMerge

Begin symbolicMerge (ExecState s_1, s_2)    //对两个状态合并
8: ζ_1 ← s_1.ζ; ζ_2 ← s_2.ζ;
9:  bool allow_merge ← (s_1.pc == s_2.pc ∧ ζ_1.size == ζ_2.size);
10: while allow_merge && ! ζ_1.empty()    //状态上下文是否一致,如果不同则不允许合并
11:    loop_info ← ζ_1.pop();
12:    if loop_info.loop_label ≠ ζ_2.pop().loop_label then
13:       allow_merge ← false;
14:    else
15:       loop_info.count ← max(ζ_2.peek().count, loop_info.count);
16:       ζ_t.push(loop_info);
17:    end if
18: end while
19: if allow_merge then    //进行合并
20:    s ← new ExecState();
21:    s.pc ← s_1.pc; s.c ← s_1.c ∨ s_2.c;
22:    s.ex ← s_1.ex ∪ s_2.ex;
23:    s.sv ← s_1.sv ∪ s_2.sv;
24:    ∀ i, s_1.sv[i] ≠ s_2.sv[i] do    //当两个状态符号值对应不同表达式时,合并成新的表达式
25:       expression ex_1 ← & s_1.sv[i];
26:       expression ex_2 ← & s_2.sv[i];
27:       s.ex.add(ex_1 ∨ ex_2);
28:       s.sv[i] ← ex_1 ∨ ex_2;
29:    end for
30:    s.cv ← s_1.cv;
31:    ∀ i, s_1.cv[i] ≠ s_2.cv[i] do    //当两个状态具体值不同时,在合并状态中将其转成符号值
32:       symbol v ← & s.cv[i];
33:       s.sv.add(v);
34:       s.ex.add(s_1.c ? v == s_1.cv[i] : v == s_2.cv[i]);
35:       s.cv.remove(i);
36:    end for
37:    while ! ζ_t.empty() do s.ζ.push(ζ_t.pop());
38:    return s;
39: else
40:    return null;
41: end if
End symbolicMerge
```

算法 4 描述优化合并函数 symbolicSetMerge 的内部逻辑,该函数的本质是将各约束通过逻辑析取进行合并,而各约束的求解则被推迟进行. 后续执行过程中的条件约束将与当前并的析取范式进行联立,并在求解时自动判断具备可满足性的命题项,因此,联立后大量多余的、或者不具备可满足性的命题项会被省略,并且不会占用推导和求解的计算资源. 函数 symbolicSetMerge 使执行状态集 S 中的各状态连同其约束尽可能地合并,从而大大减少我们在算法中实际需要处理的状态与约束数量. 它通过调用 symbolicMerge 两两合并执行状态(第 2 行). 在函数 symbolicMerge 中,算法比较两个状态的执行上下文(循环的嵌套信息)是否一致,如果不一致则不允许合并(9 ~ 18 行). 状态合并时由于其执行位置 pc 一致,会自动合并它们的符号操作表达式 ex(第 22 行)、析取符号约束 c(第 21 行)等. 当这两个状态的符号值 sv 一致时,则将符号值直接并入合并后的执行状态(第 23 行),否则将符号值对应的表达式析取合并,写入到合并后的执行状态中(24 ~ 29 行). 而

对于两个状态非符号变量的具体值 cv,算法会做类似处理. 当它们一致时,直接将该具体值写入合并后的状态(第 30 行),否则会将其转成类似同号表达式的符号值形式(31 ~36 行),以符号变量存入合并后的状态.

我们仍以图 1(a)的代码为例,经 3 次迭代后,$\Psi(l)$最终被更新为 $<\varnothing,\{s_2,s_3,s_4,s_5\}>$. 如果不进行合并优化,状态 s_2,s_3,s_4,s_5 要分别对循环 l 的后续路径做符号推导,效率低下. 因此,算法 5 对这 4 个状态做合并优化. 由于这 4 个状态都处于循环的出口,位置 pc 是一致的. 合并后的约束变成 $x\leqslant 4$ 且变量 a 会由具体变量转为符号变量. 其余变量和表达式的值由于在各个合并前的状态中是一致的,故而合并之后并不发生变化.

3.4 相关细节讨论

本节讨论一些算法的实现细节. 从理论上来说,当符号执行的中的每一个条件约束都能精确求解时,本文方法能得到程序中每个循环的准确边界值. 然而由于约束求解器的限制,有些含有复杂符号约束分量(例如非线性约束)的程序不能完整求解. 在这样的状况下,本文算法的实现会忽略不可解的约束分量,但对可解的约束分量进行多次求解采样,并由生成的测试用例对程序进行测试来获得循环边界的近似值,即循环内界. 具体来说,在算法 4 的第 10 行不能精确求解时,算法会直接制导执行状态 s 离开当前循环,并依据可解部分的符号约束生成多个测试用例. 例如:某循环在第 1000 次迭代时的路径条件约束仍然可解,但进入第 1001 次迭代时,由于循环体对符号值的更新而引入了复杂约束分量,从而导致符号执行平台无法精确求解,这时候本文的算法实现会求解可解的路径条件约束,保证产生的每个测试用例能使该循环迭代 1000 次以上,并由生成的多个测试用例来执行程序,以实际执行中该循环达到的最大迭代次数作为循环内界. 这样处理方式还能保证产生的循环边界一定具有可达性,即保证本文算法产生的循环边界一定是循环内界.

在传统符号执行中,状态的合并往往会因为约束条件变复杂而大大增加约束求解的复杂度. 然而本文的状态合并仅针对循环内界问题的求解,其状态的合并与求解具有一定的规律性. 设某循环边界值为 n,执行状态 s 到达该循环的前置条件为 Pre. 为了表述上的方便,我们将使状态 s 在经过 i 次($i\leqslant n$)迭代后离开循环的循环条件约束记为 p_i. 事实上,本文内界分析中真正需要求解的约束仅为达到 n 次迭代的条件约束 $Pre\wedge p_n$,以及经过状态合并后循环的整体约束 $Pre\wedge(p_0\vee p_1\vee p_2\vee\cdots\vee p_{n-1}\vee p_n)\wedge Post$. 这里的约束分量 $Post$ 特指状态 s 在执行完当前循环后进一步后续执行的后置约束条件. 求解 $Pre\wedge p_n$ 的目的在于获得达到当前循环上界的测试输入,这一约束的求解与状态合并无关. 而真正可能引起约束求解变复杂的部分在于求解 $Pre\wedge(p_0\vee p_1\vee p_2\vee\cdots\vee p_{n-1}\vee p_n)\wedge Post$,其目的在于分析当前循环的后置循环. 本文的状态合并,实质上使得原本被分散在 n 个执行状态中的 $Pre\wedge p_0\wedge Post$、$Pre\wedge p_1\wedge Post$、…、$Pre\wedge p_n\wedge Post$ 被合并成了 $Pre\wedge(p_0\vee p_1\vee p_2\vee\cdots\vee p_{n-1}\vee p_n)\wedge Post$,这样做大大减少了 n 个状态分别进行约束收集所占用的计算时间和存储空间,并在最终进行统一的约束求解. 由 SMT 可满足性求解的性质可知,当某一个分量 $Pre\wedge p_i\wedge Post$ ($0\leqslant i\leqslant n$)存在解时,该解也必满足合并后的约束条件. 因此,求解合并后的状态约束的过程实质上是在分量 $p_0\vee p_1\vee p_2\vee\cdots\vee p_{n-1}\vee p_n$ 中尽早搜索到一个分量 p_i,并解得 $Pre\wedge p_i\wedge Post$ 可满足性解的过程. 在求解搜索的最坏情况下,会把 n 个约束分量完整遍历一遍,其求解复杂度和不进行状态合并时分别求解 n 个状态的复杂度完全相同. 由此可见,在本文的循环内界分析这一特定问题上,状态合并不会增加符号执行的求解复杂度,而会大大加快路径的执行效率和约束的收集效率.

4 实验评估

我们在学术界常用的基准用例程序[2,3]上对本文提出的循环内界分析方法进行了实验评估. 这里的基准用例程序主要包括:Mälardalen WCET 程序集[15],Debie 用例程序[16],以及 Scimark2.0 用例程序[17]. Mälardalen WCET 程序集由瑞典 Mälardalen 大学收集构建,共包含 12142 行代码、152 个循环. Debie 用例程序是由欧洲航天局于 2001 年成功发射进入太空的 PROBA-1 卫星[18]板载系统构建的一个公开子系统. 它由多个实时任务组成,包含 13958 行代码和 75 个循环. Scimark2.0 是一组通过科学和数值计算来度量多核性能的基准程序[17]. 因为本文方法是用 C 语言实现,所以我们只评估了 Scimark 程序中的 C 语言版本,共包含 1231 行代码和 34 个循环.

4.1 方法实现和实验设置

本文基于当前主流的符号执行平台 KLEE[9]实现了符号执行制导的循环内界分析. 由于 KLEE 运行于 LLVM 编译平台,在 LLVM 中间码(LLVM IR)上生成执行状态,因此本文另外实现了一个源代码级的预处理器,用来同步源代码和 LLVM IR 之间的循环信息. 该预处理器基于 ROSE 静态分析框架[19]实现,它会在源代码里检测循环结构,并将循环的同步信息(各循环的入口、出口条件)标记到 LLVM 中间码上,从而使符号执行的制导能迅速定位到循环,并完成循环内界分析. 图 3 给出了本文的实现框架,这里我们使用 clang 作为 LLVM 编译器,在 Dell Inspiron 15R 平台上进行了实验,

平台处理器型号为 CPU Intel(R) Core(R) i7－3612QM (4 cores,2.1GHz),8G 内存,操作系统是 Ubuntu 14.04 LTS,Linux 内核版本为 3.13.0－46－generic.

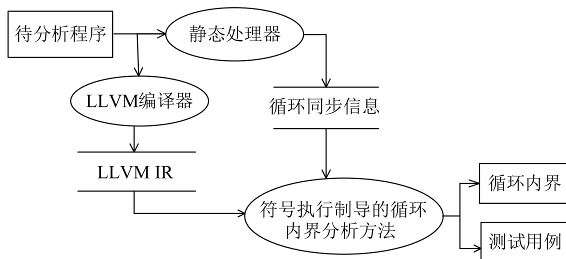

图3　循环内界分析的数据流图

4.2　分析能力评估

在实验中,我们对比了已有的循环边界分析方法 TuBound[20] 和 r-TuBund[3],这两种方法基于静态程序分析来获得循环边界.由于程序逻辑的复杂性,理论上准确获得任意循环边界是一个不可判定问题,因此,已有的方法和本文的方法都仅针对程序中常见的循环逻辑分析其边界.表 1 展示了各方法在实验用例中成功完成准确分析的循环数.从表 1 可以看出,本文的符号执行制导方法准确完成了测试程序中 261 个循环中 237 个循环的边界分析,而 TuBound[20] 和 r-TuBund[3] 分别成功完成了 202 与 206 个循环的边界分析.这一结果表明,本文的循环内界分析方法在分析能力上高于传统的静态分析方法,其主要原因在于静态分析很难处理程序源代码中因一些较为复杂机制而带来的程序歧义.例如,在程序静态分析中,程序指针和多态等源代码结构很容易导致分析不够精确,而符号执行由于仅推导符号相关的约束项,可以很好地解决这一问题.经过进一步分析,实验中有 24 个循环因以下 2 种原因未能由本文方法获得准确边界:(1)死循环或无限循环(例如 Debie 程序在 measure.c 源代码第 309 行的循环);(2)约束过于复杂的循环(例如 Scimark 程序在 kernel.c 源代码第 25 行的循环).对于死循环,由于其本身并不存在理论循环边界,可以通过预处理过滤的方式跳过这些循环的边界分析.而对于约束过于复杂的循环,则有待于符号执行约束求解能力的进一步提升.

表 1　不同方法准确分析循环边界的循环数量比较

评估对象	总循环数	TuBound	r-TuBound	内界分析	LOC
Mälardalen	152	120	121	147	12142
Debie	75	58	59	61	13958
Scimark	34	24	26	29	1231
Total	261	202	206	237	27331

4.3　分析精度评估

当循环内界分析不能准确获得循环的理论边界时,会产生略小于理论边界的近似值,以及使循环能达到该近似值的程序测试用例输入.分析精度描述了这样的近似值是否足够接近于循环的理论边界,表 2 通过准确分析循环数占循环总数的百分比和平均边界相对误差分别从分析数量上和准确度上度量了本文方法的分析精度,并具体给出了各评估对象中边界最大循环的理论边界值,以及边界最大循环经本文方法分析所得的循环内界值.当 b 为理论循环边界,b' 为本文方法分析得到的循环内界时,单个循环的边界相对误差由公式 $|b'-b|/b*100\%$ 计算.表 2 中给出了各评估对象中所有循环的平均边界相对误差.这里我们采用了人工标注的方法来获得实验用例程序中每一个循环的理论边界值.具体实施方式如下:我们将用例程序中的每一个循环随机交给两位软件工程师进行独立标注,当两者的标注结果完全一致时,其标注结果即作为该循环的标注边界,否则我们会将两位工程师的标注交给第三位工程师进行裁决,来获得标注边界.最终这些标注边界会由本文作者独立进行进一步的校对,以保证得到可信的理论边界.对于较为复杂的循环,为了防止理论边界值产生偏差,或倾向实验中的某一种自动化方法,我们主要采用人工理论推导来完成理论边界的计算(例如通过人工推导循环不变式与递推式),并由人工推导所得的测试用例来进行验证,从而保证理论边界这一评价指标的独立性与客观性.在实验中,为了防止自动分析过程因处理死循环而影响分析效率,我们将 5000000 设为循环内界分析的上限,即当所处理的循环边界到 5000000 以上时,分析程序会直接求解当前的路径约束而跳过该循环的分析,以便分析程序能够尽快覆盖到程序中的其它循环.从表中的数据可知,Scimark 用例程序中因存在超过上限 5000000 的高边界循环,因而获得的最大内界分析值不能达到理论边界值.在统计分析精度时,我们未将这些不能完成边界分析的循环计算在内.从所得的准确分析循环数占比和平均边界相对误差来看,本文的循环内界分析精度较高,各用例程序的平均边界相对误差都小于 1%,且完全精确分析边界的循环数占比在80%以上.产生误差

表 2　循环内界的分析精度

评估对象	人工标注的最大理论边界值	分析获得的最大内界	准确分析循环数	准确分析循环数占循环总数百分比	平均边界相对误差
Mälardalen	8990	8990	147	96.71%	0.07%
Debie	17676	17676	61	81.33%	0.98%
Scimark	67108864	≥5000000	29	85.29%	0.71%

的主要原因是 SMT 约束求解器会低估条件复杂的循环,因此,随着符号执行约束求解能力的发展,本文方法的分析精度仍会进一步得到提高.

4.4 分析效率评估

本文循环内界分析方法实现了状态约束的优化合并,在实验中我们进一步分析了优化对于分析效率的影响,表 3 描述了使用本文符号执行制导的循环内界分析策略与不带优化的传统符号执行策略(KLEE 默认执行策略)在执行效率上的对比情况. 在传统符号执行策略中,我们以第一个执行状态离开循环的时间作为它的循环分析时间. 尽管这样的设定会高估 KLEE 默认执行策略的执行效率,但从表 3 的数据来看,经过状态约束优化合并制导策略的内界分析方法仍然比传统的符号执行策略在执行效率上要高出 140% 以上. 结合表 1 和表 3 的数据,本文方法的处理能力大约为每分钟 10000 行代码,这能够符合大多数应用的要求.

表 3 平均执行时间(同传统符号执行 KLEE 相比)

评估对象	KLEE 执行循环数	KLEE 平均分析时间 t_1(s)	内界分析平均时间 t_2(s)	速度提升 $(t_1-t_2)/t_2$
Mälardalen	107	121.77	46.21	163.51%
Debie	48	92.29	23.12	299.28%
Scimark	17	170.39	68.85	147.46%

5 相关工作

本节介绍近年来与本文方法紧密联系的相关研究,主要包括:(1)循环边界分析研究;(2)与循环相关的符号执行制导策略研究.

循环边界分析 循环边界分析是一类重要的程序分析方法,它计算循环在程序不同执行状况下的最大迭代次数. 已有的研究主要通过静态分析源代码和二进制码来计算循环边界. Healy 等学者提出了基于手工标注的循环边界推导方法,该方法需要用户标注循环相关变量的取值范围,并由这些标注信息推导循环边界[21]. Michiel 等学者通过静态分析控制流并对循环条件和相关操作进行抽象,从而保守分析 C 程序的循环边界[1]. Gulwani 等学者基于抽象解释的线性不变量生成来分析循环边界[22]. 此外,他们还进一步改进了该方法的抽象解释过程,提出了一种基于规则证明的技术来处理计算循环边界时所需的不变量析取式[23]. Henzinger 等学者开发的循环分析软件[24]通过代数计算系统(Mathematica)来自动求解循环的边界断言并自动推导循环不变量. 这些方法均通过静态分析技术来保守计算循环外界,以确保循环边界的安全性. 与这些方法不同,本文方法基于符号执行技术计算程序的循环内界,以便用户对循环边界的分析结果具有更为完整的认识.

符号执行制导 符号执行以符号推导来分析软件在任意输入下的执行状态. 传统的符号执行技术关注于软件测试用例的生成,但由于循环结构本身是符号执行中较难处理的问题,因此近年来出现了很多与循环相关的符号执行制导策略研究. Godefroid 提出了一种符号执行制导策略[25],称为 SMART 策略,它通过模块化的思想将符号执行的推导流程模块化,从而大大提高了符号执行的制导效率. 更进一步,它们给出了称为 SAGE[12] 的扩展算法,它通过多种方式对路径记分,并依据分值来制导符号执行的搜索. Xie 等学者提出了一种基于拟合度函数的符号执行策略,对于不同的搜索分支,该方法通过计算由多个拟合度函数组合而成的拟合度值来制导符号执行过程[14]. Cadar 等学者编写了软件工具 EXE[11],并将其扩展成 KLEE[9]. 这些工具均使用了深度优先搜索配合启发式的方法来制导符号执行. Li 等人提出了基于子路径制导的搜索策略[26],该策略分析已覆盖路径中长度为 n 的子路径,然后决定哪些执行路径会被最少遍历. Xiao 等学者[7] 总结了关于利用符号执行来处理循环的研究工作,这些工作[27,28] 主要通过静态循环摘要的方法使符号执行不会在循环处理上逗留过长的时间. 在此之后,Xie 等学者编写了工具 S-Looper[29],该工具进一步提供了摘要处理字符串循环的方法. 同这些已有的研究不同,本文关注利用符号执行来分析循环内界,并同时优化了符号执行处理循环的能力.

6 总结

循环是计算机中重要且较为复杂的程序结构. 许多应用场景需要在静态条件下计算循环所能达到的最大迭代次数,即循环边界. 现有的循环边界分析技术均采用保守策略,强调边界分析的安全性. 在不能准确分析边界精确值时,保守策略会保证分析值一定超出实际边界值,故而得到的分析结果为循环外界. 本文提出了一种符号执行制导的方法来分析循环的内界,即保证循环边界的分析值一定能被测试用例覆盖到,并同时产生覆盖该内界的测试用例. 当用户结合循环外界分析和循环内界分析时,可以准确地获得循环实际边界的范围区间,从而对分析结果有更为完整的认识.

本文通过制导符号执行来实现循环内界分析. 符号执行将程序输入定义为可代表任意值的符号,并推导程序在任意输入下的执行状态. 通过制导符号执行的路径搜索,优化合并状态约束,本文提出了完整的循环内界分析算法. 我们基于开源符号执行平台 KLEE 实现了内界分析方法,并在多组已有研究采用的基准用

例集上进行了实例评估,实验结果显示本文循环内界分析方法具有准确性和高效性.

参考文献

[1] Michiel M D, Bonenfant A, Casse H, et al. Static loop bound analysis of C programs based on flow analysis and abstract interpretation[A]. Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications[C]. Taiwan:IEEE Computer Society,2008. 161 – 166.

[2] Gustafsson J, Ermedahl A, Sandberg C, et al. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution[A]. Proceedings of the 27th IEEE International Real-Time Systems Symposium [C]. Rio de Janeiro:IEEE,2006. 57 – 66.

[3] Knoop J, Kovacs L, et al. Symbolic loop bound computation for WCET analysis[A]. Proceedings of the 8th International Conference on Perspectives of System Informatics [C]. Berlin:Springer Verlag,2011. 227 – 242.

[4] Wilhelm R, Engblom J, Ermedahl A, et al. The worst-case execution-time problem—overview of methods and survey of tools[J]. ACM Transactions in Embedded Computing Systems,2008,7(3):1 – 53.

[5] Cadar C, Godefroid P, Khurshid S, et al. Symbolic execution for software testing in practice[A]. Proceedings of the 33rd International Conference on Software Engineering [C]. New York:ACM,2011. 1066 – 1071.

[6] King J C. Symbolic execution and program testing[J]. Communications of the ACM,1976,19(7):385 – 394.

[7] Xiao X, Li S, Xie T, et al. Characteristic studies of loop problems for structural test generation via symbolic execution[A]. Proceedings of the 28th International Conference on Automated Software Engineering [C]. California, IEEE,2013. 246 – 256.

[8] Anand S, Pǒsǒreanu C S, Visser W. JPF – SE:A symbolic execution extension to java path finder[A]. Proceedings of the 13th International Conferrence on Tools and Algorithms for the Construction and Analysis of Systems[C]. Berlin:Springer-Verlag,2007. 134 – 138.

[9] Cadar C, Dunbar D, Engler D. KLEE:Unassisted and automatic generation of high-coverage tests for complex systems programs[A]. Proceedings of the 8th USENIX Conference on Operating systems design and implementation [C]. California:USENIX Association,2008. 209 – 224.

[10] Boonstoppel P, Cadar C, Engler D. RWset:Attacking path explosion in constraint-based test generation[A]. Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems[C]. Berlin:Springer-Verlag,2008. 351 – 366.

[11] Cadar C, Ganesh V, Pawlowski P M, et al. EXE:automatically generating inputs of death[A]. Proceedings of the 13th ACM conference on Computer and communications security[C]. New York:ACM,2006. 322 – 335.

[12] Godefroid P, Levin M Y, Molnar D. Automated whitebox fuzz testing[A]. Proceedings of the 16th Annual Network and Distributed System Security Symposium[C]. California:Internet Society,2008.

[13] Majumdar R, Sen K. Hybrid concolic testing[A]. Proceedings of the 29th International Conference on Software Engineering[C]. Washington DC:IEEE Computer Society,2007. 416 – 426.

[14] Xie T, Tillmann N, Halleux J D, et al. Fitness-guided path exploration in dynamic symbolic execution[A]. Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks[C]. Lisbon:IEEE,2009. 359 – 368.

[15] Gustafsson J, Betts A, Ermedahl A, et al. The mälardalen WCET benchmarks:Past, present and future[A]. Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis[C]. Dagstuhl:Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik,2010. 136 – 146.

[16] Holsti N, Gustafsson J, Bernat G, et al. WCET 2008-Report from the Tool Challenge 2008-8th Intl. workshop on worst-case execution time (WCET) analysis[A]. Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis[C]. Dagstuhl:Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik,2008. 1 – 23

[17] Pozo R, Miller B. SciMark:A numerical benchmark for Java and C/C++[J/OL]. http://math.nist.gov/scimark2/,2004 – 03 – 31.

[18] Kuitunen J, Drolshagen G, Mcdonnell J, et al. DEBIE-first standard in-situ debris monitoring instrument[A]. Proceedings of the Third European Conference on Space Debris[C]. Netherlands:ESA Publications Division,2001. 185 – 190.

[19] Quinlan D. ROSE:Compiler support for object-oriented frameworks[J]. Parallel Processing Letters,2000,10(0):215 – 226.

[20] Prantl A, Knoop J, Schordan M, et al. Constraint solving for high-level WCET analysis[R]. Udine:Computing Research Repository,2009. 77 – 89.

[21] Healy C, Sjodin M, Rustagi V, et al. Supporting timing analysis by automatic bounding of loopIterations[J]. Real-Time Systems,2000,18(2):129 – 156.

[22] Gulwani S, Mehra K K, Chilimbi T, et al. Speed:Precise and efficient static estimation of program computational complexity[A]. Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Pro-

gramming Languages[C]. New York:ACM,2009. 127 –139.

[23] Gulwani S,Zuleger F. The reachability-bound problem [A]. Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation[C]. New York:ACM,2010. 292 –304.

[24] Henzinger T A,Hottelier T,Kovács L. Valigator:A verification tool with bound and invariant generation[A]. Proceedings of the 15th International Conference on Logic for Programming,Artificial Intelligence,and Reasoning[C]. Berlin:Springer-Verlag,2008. 333 –342.

[25] Godefroid P. Compositional dynamic test generation[A]. Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Aymposium on Principles of Programming Languages [C]. New York:ACM,2007. 47 –54.

[26] Li Y,Su Z,Wang L,et al. Steering symbolic execution to less traveled paths[A]. Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications[C]. New York:ACM,2013. 19 –32.

[27] Godefroid P,Luchaup D. Automatic partial loop summarization in dynamic test generation[A]. Proceedings of the 2011 International Symposium on Software Testing and Analysis[C]. New York:ACM,2011. 23 –33.

[28] Saxena P,Poosankam P,Mccamant S,et al. Loop-extended symbolic execution on binary programs[A]. Proceedings of the Eighteenth International Symposium on Software Testing and Analysis[C]. New York:ACM,2009. 225 –236.

[29] Xie X,Liu Y,Le W,et al. S-looper:automatic summarization for multipath string loops[A]. Proceedings of the 2015 International Symposium on Software Testing and Analysis[C]. New York:ACM,2015. 188 –198.

作者简介

赵祖威 男,1992 年 10 月出生,江苏南通人. 2015 年本科毕业于南京大学软件学院. 现为南京大学软件学院硕士研究生,主要研究方向为静态程序分析,软件测试.

汤恩义(通信作者) 男,1982 年 9 月出生,江苏苏州人,博士. 现为南京大学软件学院助理研究员,主要研究领域为软件工程,新型软件测试方法,数值计算,程序分析方法.

E-mail:eytang@ nju. edu. cn

冯世宁 女,1986 年 11 月出生,江苏南京人. 2008 年、2011 年分别在东南大学、国网电力科学研究院获学士、硕士学位. 现为南瑞集团(国网电力科学研究院)工程师,从事系统仿真,电力电子自动化等领域的研究.